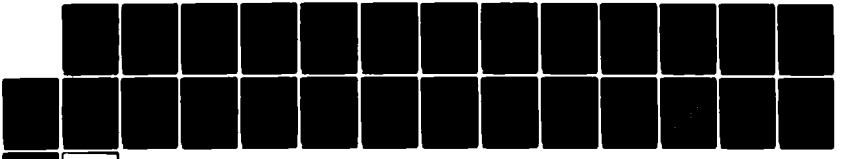
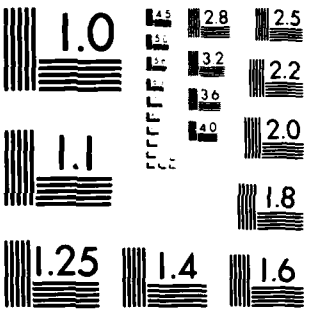
AD-A144 384 CONCURRENT ACCESS TO DATA STRUCTURES(U) INSTITUT FUER 1/1
INFORMATIK ZURICH (SWITZERLAND) J NIEVERGELT SEP 82
DAJA37-82-C-0058

UNCLASSIFIED F/G 9/2 NL



END
DATE FILMED
9 84
DTIC



Quarterly Progress Report #2, September 1982

ERO Contract No. DAJA 37-82-C-0058

Concurrent access to data structures

Principal Investigator:

Prof. Jurg Nievergelt,
Informatik, ETH,
CH-8092 Zurich, Switzerland

CONCURRENT ACCESS CONTROL IN 2-LEVEL FILE STRUCTURES

H. Hinterberger and J. Nievergelt
Informatik, ETH, CH-8092 Zurich

Preliminary draft - all comments welcome

Table of contents

Abstract

File structures designed for modern data base systems must allow a high degree of concurrent access. This holds for distributed as well as centralized data bases. Ideally, two processes that access disjoint data are allowed to proceed simultaneously, unhampered by the other's presence. However, such processes may interfere due to access path collition, for example at the root of a tree structured file. Elaborate and costly protocols have been developed to guarantee correct operation in concurrently accessed B-trees.

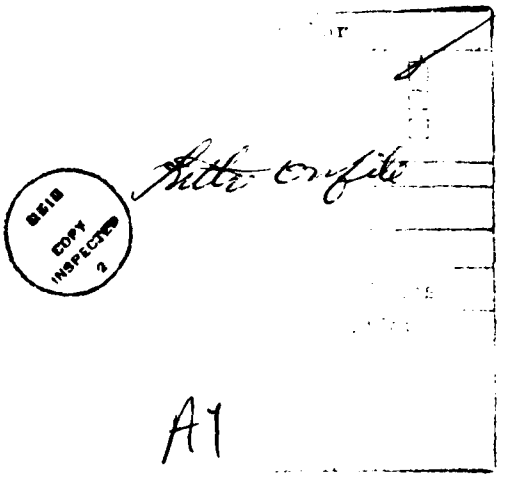
We show that 2-level file structures allow simpler concurrent access protocols, and present a control scheme that strikes a good compromise between the conflicting goals of simplicity and maximal concurrency. We propose a definition of what it means for a file system to be correct and show that the simple and efficient protocol designed is correct.

Index terms

Data base, file structures, extendible hash files, concurrent access, concurrent programming.

Acknowledgement

We are grateful to Carla Ellis and Jay Misra for discussions that stimulated our interest in concurrent access to data structures.



Supported in part by the European Research Office, US Army, under contract DAJA37-82-C-00589, R&D 4093-CC.

1. The scope of concurrency problems

1.1 Historical development

When operating systems started to support multiprogramming, explicit attention had to be given to the problem of concurrency. Some of the terms gaining importance in connection with concurrency were: Process, Processor, Dispatching, Virtual CPU, Monitor, Time slicing etc. If several programs are active simultaneously in a comuter system, questions of synchronization arise. Work on synchronization problems generated more terminology: Deadlock, Mutual Exclusion, Readers & Writers, Preemption etc. became concepts that helped in the formalization of these problems. Synchronization primitives such as test & set, wait & signal, semaphores and their associated operations were introduced. More recently, attention has been focused on problems of concurrency in connection with distributed systems (distr. software, distr. data bases etc.). The problems that need to be studied in distributed systems are more complex than the problems dealt with in operating system design. Notions such as Transaction Management (Transactions consisting of many steps), Consistency and Integrity, Serializability, problems of Lost Updates etc. became issues requiring clean solutions. Problems dealing with security, crash recovery, legal scheduling are complex and not as easy to formalize as the "toy problems" of the first phase in the development of concurrency.(e.g. producer-consumer, dining philosophers [HGLS 78]). The various methods for concurrency control proposed so far fall into two general classes: 1) time stamp ordering of actions and 2) locking of granules. With time stamps, the transactions are ordered either at starting time or commit time. The ordering can thus be controlled by the user. When locks are employed, a two phase protocol is usually implemented (maximum locking before unlocking) to order transactions. Control in this case rests with the system and cannot be influenced by the user. The reader is referred to [GM 82] for a comparison of the two methods.

We are particularly interested, among the many problems raised by concurrency, in concurrent access to a file system. A problem of intermediate complexity and generality (on the scale of paragraph 1.1). It is significantly more complex than the interaction of just two different processes, such as producer-consumer for example, or the synchronization of an arbitrary number of identical processes, such as in mutual exclusion. On the other hand, our problem is simpler than, for example, consistency questions that may involve any number of processes about whose purposes and actions we know nothing, except that they access certain entities. Our problem involves an arbitrary number of different processes whose semantics are precisely known. We consider a 2-level file system for managing a collection of records and the operations FIND, INSERT and DELETE operating on these individual records.

1.2 Concurrency and file systems

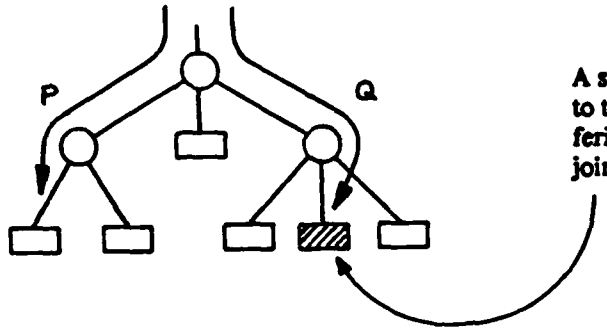
Various data structures have been proposed for the implementation of a file system that provides direct access (FIND) and modification (INSERT, DELETE) of a single record at a time. Among the most efficient ones are tree structures and address computation techniques. Textbooks on data structures, e.g. [Knu 73], typically consider these data structures only in a sequential environment. Exclusive access to the entire structure is given to one process at a time (serialization of processes). This assumption is often reasonable for data collections that can be stored entirely in central memory; but is quite inappropriate for data base or transaction processing systems of realistic size.

Furthermore, since the more efficient data structures tend to require occasional rebalancing operations which are quite time consuming (several disk accesses), the mutual exclusion of processes becomes even more inappropriate. A rebalancing operation is usually restricted to a local area of the data structure but in the worst case can propagate to remote areas. It is inefficient to lock the entire collection of data for a process that changes some pointers. Structural changes in one part of the data collection should not hold up processes that access a disjoint part of this same data collection. Indeed, the following principle of efficient implementation of concurrent access to data collections is desirable:

> If two processes, P and Q, access data located in different units of storage (e.g. pages or buckets), then, P and Q should be able to proceed unhampered by each other.

This principle can certainly be realized: for example in a random access memory, where each possible key value is assigned a different (and unique) storage unit. It cannot be easily realized in data structures

that pack records tightly and make them accessible by linked directories. There seems to be a trade-off between utilization and permissible concurrency: The more the records are spread out, the more records are accessible simultaneously; the tighter they are packed, the more synchronization (and serialization) constraints have to be introduced. This intuitive conclusion is not surprising. The purpose of this paper is to show that, even among data structures that are roughly equivalent in their storage utilization and sequential access efficiency, there are great differences in concurrent access efficiency. Specifically, several papers provide evidence that concurrency in tree structures require elaborate protocols. This is due to the simple fact that every node in a tree is the entry point to the entire subtree rooted at this node and consequently is a potential bottleneck. Consider the paths of processes P and Q in the following picture.



A split or merge here may propagate to the root of the tree, thus interfering with process P accessing disjoint data.

See [BS 77], [KL 78], [MS 78], [Ell 79] and [Ell 80] for discussions on concurrent access to tree structures.

We show that address computation techniques, which are devoid of bottlenecks such as the root of a tree (or subtree), allow significantly more concurrency and require simpler protocols than do tree structures. We will not differentiate between centralized and distributed data bases. In both cases a file system is required and it is the control of this file system that concerns us here.

2. 2-level file structures

2.1. 2-level memory systems

The two points to be considered first when discussing memory systems are:

1) There are no very fast, extremely cheap, reliable memories available yet.

2) For various reasons (interactivity, long term storage, etc.) a per bit cost trade-off has to be made.

A reduced memory cost per bit and expensive software favor the use of larger memories. If the fastest storage device were also the cheapest and most reliable, then one would obviously employ only a single level of memory. This is not so, however, and today practically all computing systems include some form of storage hierarchy. Improvements in technology have still not changed the fact that fast memories are much more expensive than slow ones. Consequently, it is most cost-effective to have the smallest, shortest access-time memory located physically close to the processor and larger, slower ones farther away, accessible via buses or channels. See [PS 81] for a survey of memory systems. This combination of expensive, high performance devices with inexpensive, lower-performance devices results in an implementation commonly referred to as a hierarchical or multi-level storage system.

Of these multi-level storage systems, the 2-level storage system incorporating a fast (20-200 ns access plus transfer time) semiconductor-based primary memory and a slower (20-100 ms) secondary memory (usually moving head disks) have found widest distribution. Naturally, one wants to supply the necessary information to the processor at the speed of the fastest memory. If this information is kept in a large (possibly dynamic) file, then by the previous reasoning, this information has to be transferred first from the slow secondary to the fast primary store. This transfer of information is very time consuming. To make things worse, most traditional file systems do not take the 2-level structure of a given memory system into consideration. As a consequence it is very difficult to optimize access to files organized using traditional file structures.

In the following two sections we discuss two file structures, extendible hashing and the grid file, which do take the 2-level structure of memory hierarchies into consideration. This provision results in increased access efficiency as well as simplified access control when information from one file has to be supplied to more than one processor.

2.2 Brief survey of extendible hashing

Extendible hash tables promise to greatly enhance the applicability of address calculation methods to large, dynamic files [FNPS 79]. The following is a brief description of the principles involved.

The hash function h generates a very long pseudokey h(K). At any given instant however, only a certain number d of bits of this key are used. d increases or decreases as the file grows or shrinks, but only logarithmically. The core of an extendible hash file is an array of 2 to the power d pointers, called an "index of depth d". This index represents the relationship between the pseudokeys and the pages (memory blocks) containing records. The first d bits of the pseudokey h(K) are used as index to the directory and the corresponding entry in the directory points to the page containing the record with key K.

If the entire index can be stored in central memory, one disk access is sufficient to find a record with a given key value K. Should the index also be stored on disk, two disk accesses will yield the required record. The first to the correct directory entry, the second to the correct data page. Fig. 2.1 illustrates the idea. The dynamic adjustment of the file system is done as follows:

- If a page is full (i.e. the next entry would cause an overflow), the contents of the page is redistributed onto two pages, which will be considered twins.
  The assignment of records to one twin or the other is based on a certain bit of the pseudokey.
  Possibly the depth d of the index has to be increased.

- If two pages with a low record count happen to be twins, then they can be merged onto one page.
  Possibly the depth d of the index can be reduced by 1.

Fig. 2.2 depicts the changes required if the system grows.

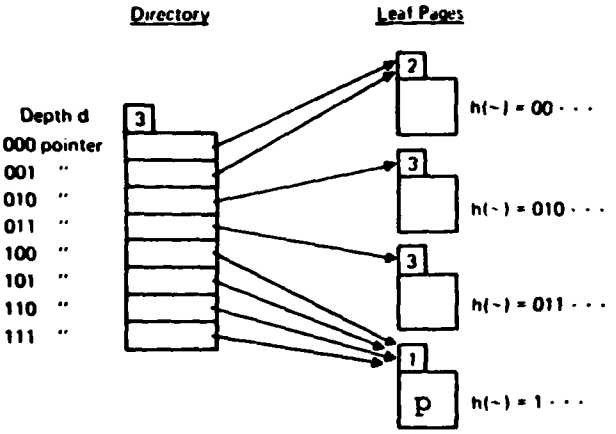


Fig. 2.1 Graphic depiction of extendible hash file organization

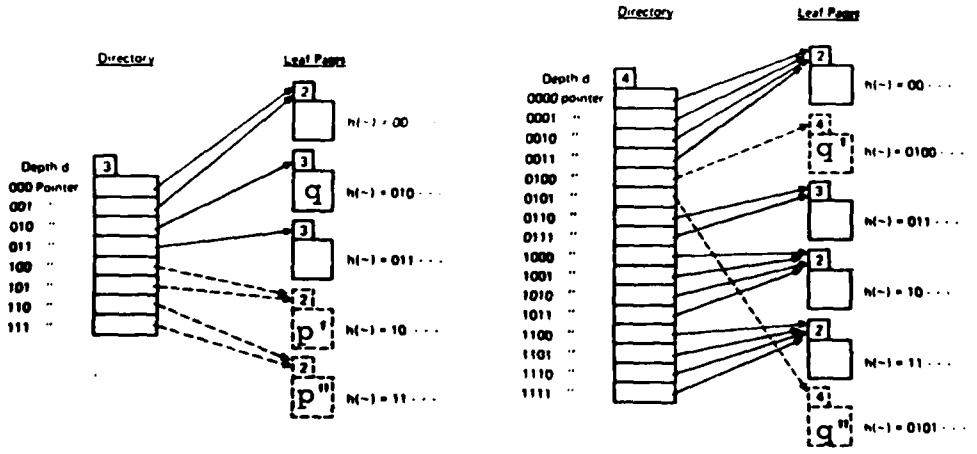


Fig. 2.2 Growth of file system: p' and p'' are the result of p (Fig. 2.1) splitting. The splitting of q into q' and q'' results in the doubling of the directory.

2.3 Brief survey of the grid file

The grid file emerged as an answer to the various deficiencies encountered when traditional file structures are used for multi-key access to dynamic files [NHS 81]. The starting point is the extreme solution given by the "bitmap representation" of the attribute space, which reserves one bit for each possible record in the space, whether it is present in the file or not. In a k-dimensional bitmap the combinatios of all possible values of k attributes are represented by a bit position in a k-dimensional matrix. Fig. 2.3 shows a 3-dimensional bitmap.

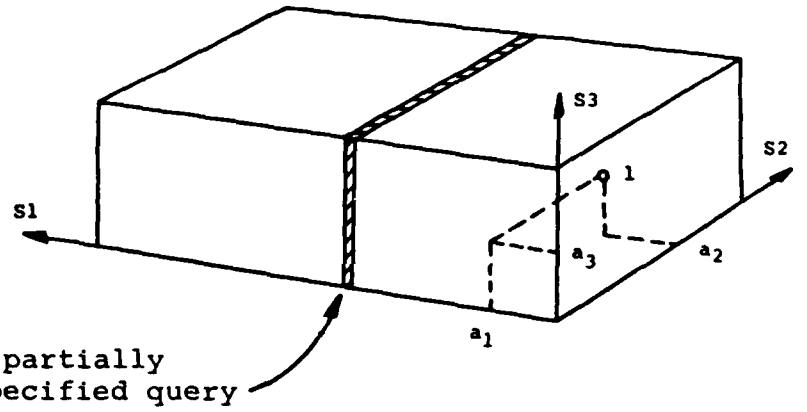


Fig. 2.3 The 3-dimensional bitmap. A "1" indicates the presence of a record with attribute values determined by its position in the map, a "0" indicates absence.

FIND(r_1,r_2,....,r_k) reduces to direct access, INSERT/DELETE requires that a position in the bitmap be set to "1" or "0" respectively, and NEXT in any dimension requires a scan until the next "1" is found. For realistic applications, this bitmap is impossibly large and has to be compressed. In maintaining a dynamic partitioning (directory) on the space of all key-values one approximates the bitmap through compression. Assuming independent attributes, but not necessarily uniform distributions, the embedding space from which the data is drawn is partitioned in a "grid-like" fashion: each region boundary cuts the entire search space in two. All attributes are given the same priority when being partitioned.

The following terminology and notation is used in the example of the 3-dimensional case. On the record space S = X x Y x Z a grid partition P = U x V x W is obtained by imposing intervals on each axis and dividing the record space into blocks, called grid blocks, as shown in Fig. 2.4.

Record space S = X x Y x Z

Grid Partition P = U x V x W

Intervals of the partition U = $(u_0,u_1,...,u_\ell)$
V = $(v_0,v_1,...,v_m)$
W = $(w_0,w_1,...,w_n)$

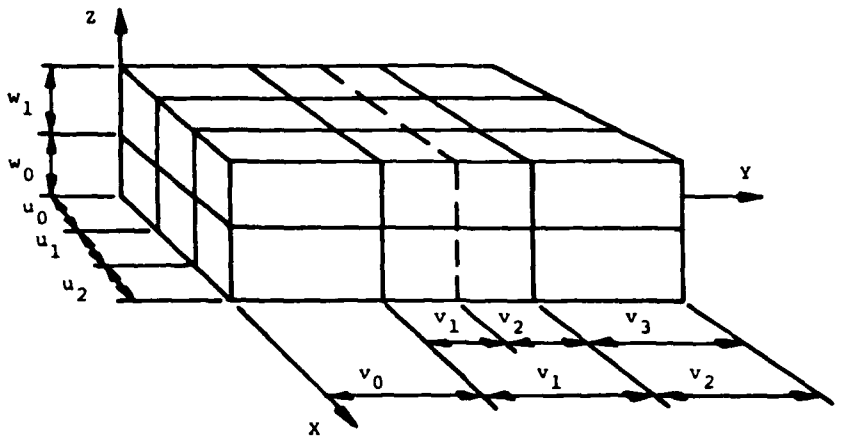


Fig. 2.4 A 3-dimensional record space X x Y x Z, with a grid partition P = U x V x W. The picture shows the effect of refining P by splitting interval v_1.

During operation of a file system the underlying partition of the search space needs to be modified in response to insertions and deletions. The grid partition P = U x V x W is modified by altering only one of its components at a time. A 1-dimensional partition is modified either by splitting one of its intervals in two, or by merging two adjacent intervals into one. Fig. 2.4 shows this for the partition V.

In order to obtain a file system, operations that relate grid blocks and records to each other are needed, such as: find the grid block in which a given record lies, or list all records in a given grid block. The data structure used to organize records within a bucket is of minor importance for the file system as a whole; the structure used to organize the set of buckets, on the other hand, is the heart of a file system. The set of buckets of a given file system is usually managed through a directory. The purpose of the grid directory is to maintain the dynamic correspondence between grid blocks in the record space and data buckets. For reasons of access efficiency, all records in one grid block must be stored in the same bucket. Several grid blocks can share a bucket (to avoid arbitrarily low bucket occupancies) as shown in Fig. 2.5. Such a set of grid blocks is called a bucket region. Bucket regions have the shape of a box, i.e. a k-dimensional rectangle. These convex regions of buckets are pairwise disjoint, together they span the space of records.

A grid directory consists of two parts: first, a dynamic k-dimensional array called the grid array; its elements (pointers to dfata buckets) are in 1:1 correspondence with the grid blocks of the partition; second, k 1-dimensional arrays called linear scales; each scale defines a partition of a domain S. The grid array is likely to be large and must be kept on disk. but the linear scales are small and can be kept in central memory. The following example illustrates how the grid file is accessed. Consider a record space with attribute "year" with domain 0 .. 2000, and attribute "initial" with domain a .. z. Assume that the distribution of records in the record space is such as to have caused the following grid partition to emerge:

Year = (0, 1000, 1500, 1750, 1875, 2000); initial = (a, f, k, p, z).

A FIND for a fully specified query $(r_1, r_2, \ldots,)$, such as FIND[1980,w], is executed as shown in Fig. 2.6. The attribute value 1980 is converted into interval index 5 through a search in scale "year", and w is converted into the interval index 4 in scale "initial". The interval indices, 5 and 4, provide direct access to the correct element of the grid directory, where the bucket address is located.

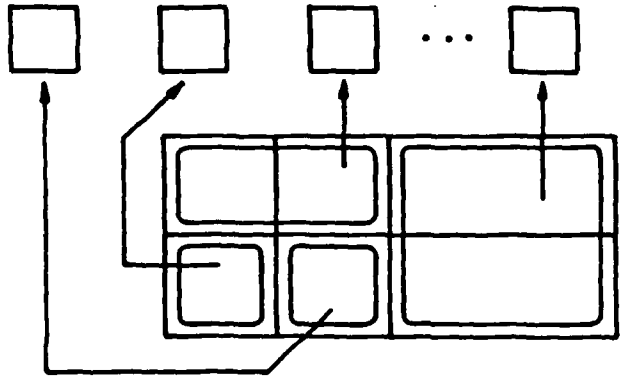


Fig. 2.5 A convex assignment of grid blocks to buckets.

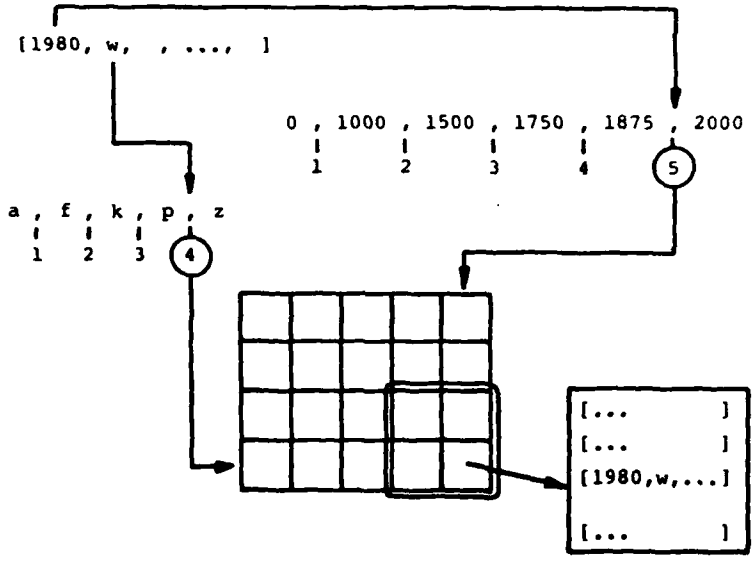


Fig. 2.6 Retrieval of a single record in the grid file.

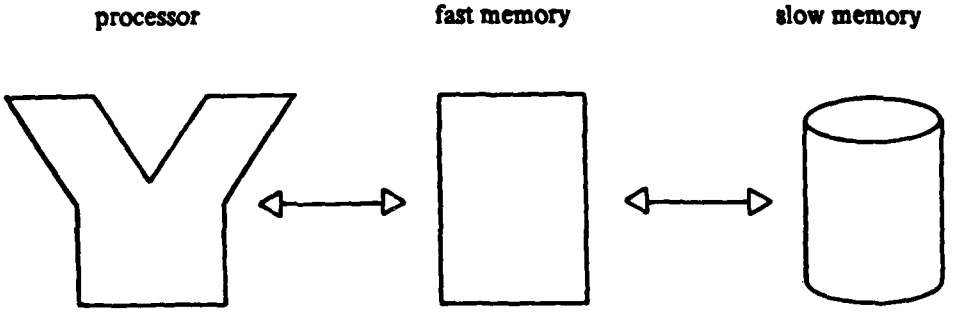
2.4 Properties of 2-level file structures

An interactive system should provide "instantaneous response" to a user's requests. This puts a heavy burden on the efficiency of a file processing system since "instantaneous = approx. 1/10th of a second = no more than 2 disk accesses. Two components affect this response time:

- time spent in waiting to access peripheral storage media,
- time spent in accessing peripheral storage media.

A given file system can greatly influence these two components. The first component ties in with the degree of concurrency the system supports whereas the second component measures the efficiency of record retrieval.

Let's recall briefly the 2-level memory system.



The maximal amount of data transferred in one access to slow memory (disk) is fixed and usually is defined to be a disk block or page. In most applications, the management of these pages is independent of the structure of the file system. This means in effect that the data stored in a given file is randomly distributed on a secondary storage medium. This is especially true for a dynamic data environment. The question then arises as to how many pages need to be transferred until a given record is located. Ideally this is one transfer per data structure. In a 1-level file structure (no indexing) this can be achieved with hashing. However, hashing destroys the order inherent in a given key (attribute) space and is difficult to apply with multi-key records. Dynamic files are also difficult to maintain when hashing is used. If, instead of key-transformation, comparative search techniques are used, then a second data structure, the directory, has to be introduced to obtain reasonable access efficiencies. In a paged system, it is sensible to use the directory to manage data on a per page basis. This directory needs to be extended (reduced) whenever pages overflow (become empty) due to changes in the underlying data. It is the intended purpose of 2-level file structures, to reduce the required structural changes both in quantity and extent.

The volume of a directory of a file of realistic size is too large to be kept in central memory. This means, that now ideally two disk accesses will fetch us a fully specified record. This is the case with extendible hashing and also with the grid file. The reason lies in the fact that in both cases the directory consists of one level only, that is, no hierarchic order exists. In both cases enough information can be processed in central memory to allow a one level directory and consequently this upper bound on page transfers. Hence the term 2-level file structure: no more than one level to store the data, no more than one level to maintain the directory.

Contrary to directories organized as trees, these 1-level directory structures provide disjoint access paths. In other words, two records located in different data buckets are accessed through different directory entries. This, together with the fact that changes in one part of the directory do not propagate to other parts, allows for increased concurrency. Fig. 2.7 depicts the different organizations.

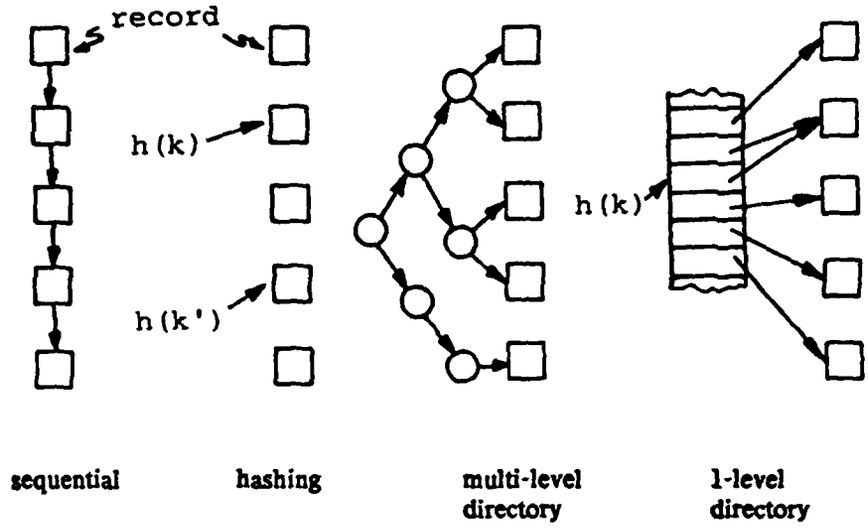


Fig. 2.7 Different file organization principles.

3. Operations on 2-level file structures

3.1 Separation of operations

In a study of concurrent processing, it is useful to start by analyzing the logical independence of the various operations that have been defined as beeing provided by a given system. Logical independence, in the sense of one operation being independent on (the result of) any other operation, and not interfering in its data access with any other operation, sets the limit of maximal potential concurrency. It is still another question whether this inherent maximal potential concurrency is worth exploiting. We investigate it in order to delimit the space of solutions to be considered.

Some data structures composed of simpler types can be operated upon either by component or as a whole. In particular, nodes can be inserted into a tree or the tree can be rotated; a record can be deleted from a hash bucket, or the keys can be rehashed. Operations that involve the data structure as a whole are triggered by operations on components that have exceeded a particular bound imposed by the structure. As an example, 2-level files provide three primary operations: FIND, INSERT, DELETE at the record level, and require four induced operations: SPLIT, MERGE, EXTEND, REDUCE, at the bucket and directory levels.

From an implementation point of view, it is reasonable to identify a bucket with a page, or other unit of data transfer between central memory and secondary storage. This mutual exclusion provides us with a simplifying framework. More interesting is the investigation of a multiprocessor environment, wih each processor competing for access to memory(ies). Here it is imossible to predict the outcome of contention and we find the above logical separation of operations not sufficient in the analysis of the effects concurrency has on the data and its structure.

It is safe to assume that, once a process is in a bucket, progress is fast. But getting to the bucket (involves accessing the directory, may or may not be in CM, as well as locating the bucket, may or may not be in CM, once we know its address) could be time consuming. Fig. 3.1 depicts schematically what we mean. All primary operations share the above characteristics.

It seems sensible to structure the primary operations taking these time constraints into consideration. This structure would also reflect the separation of the part dealing with the data structure (slow) from the part dealing with individual records (fast). Accordingly we have the following skeleton for each primary operation:

- locate-bucket
- operate within-bucket

Between these two operations there is a clear logical independence. We will therefore analyze the within-bucket and the locate-bucket operations independently.

If a user inserts a record, he does not (and should not) care whether his request caused some bucket to be split or not. Therefore one more separation of operations has to be introduced. This separation is necessary in order to make the induced operations transparent to a user of the file management system. We will call the operations issued by the system (transparent to the user) internal operations and the operations issued by the user external operations.

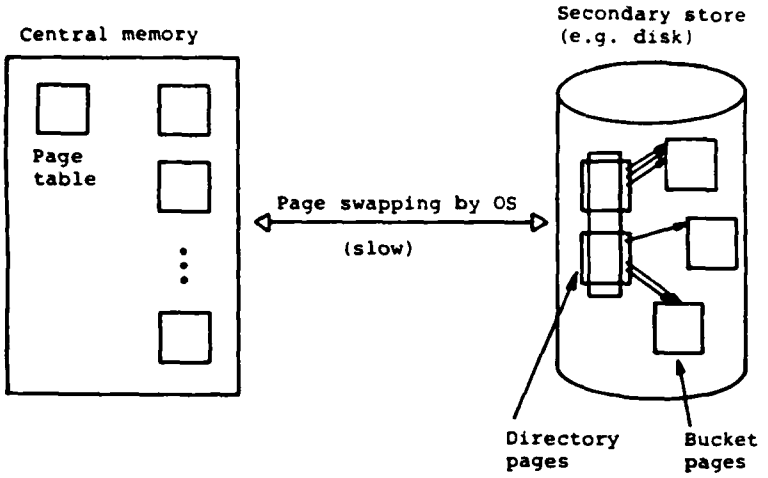


Fig. 3.1 Possible file system implementation

3.2 Internal operations

First, we list the sensitive operations involving the location of the bucket address for a given key-value z. Sensitive, because a) the parameter B for all within-bucket operations has to be determined correctly under any circumstance, b) their speed can vary from *fast (everything in CM)* to *slow* (two disk accesses). Then the induced operations are tabled.

Procedure	Outcome
locate_bucket(z):	- bucket pointer B.
split(B):	- directory OK. - directory sparse.
merge(B):	- directory OK. - directory redundant.
extend:	- within limit of granularity. - limit of granularity reached.
reduce:	- possible. - attribute eliminated.

Each of the three primary operations (FIND, INSERT, DELETE) has its within-bucket counterpart. Two parameters are passed to each of these procedures. They are the key-value z and the bucket address B. These procedures return values as defined below.

Procedure	Outcome
find_within_bucket(z,B)	- found. - not present.
insert_within_bucket(z,B)	- already present. - inserted. - split request.
delete_within_bucket(z,B)	- not present. - deleted. - merge request.

3.3 External operations

The external procedures provided by the file system are synthesized from the internal procedures of the previous paragraph as follows:

External Procedure	Implementation
FIND(z)	1. locate_bucket(z) 2. find_within_bucket(z,B)
INSERT(z)	1. locate_bucket(z) 2. insert_within_bucket(z,B) 3. split(B) 4. extend
DELETE(z)	1. locate_bucket(z) 2. delete_within_bucket(z,B) 3. merge(B) 4. reduce

INSERT and DELETE terminate after a successful outcome of step 2, 3 or 4 respectively. If the directory cannot be extended or reduced, a warning is issued.

4. Requirements on a file system that supports concurrent access

By calling our system a file system, we stress the fact that we are primarily concerned with the correct handling of records (as opposed to transactions) in a concurrent environment. In other words, we defer the enforcement of consistency constraints imposed on the data base, scheduling, fairness etc. to another implementation level, say a DB-management system or a transaction manager.

One difficulty generally encountered in asynchronous concurrent processing is the maintenance of a valid structure of the data base. This problem has been studied mostly for tree structures (see e.g. [BS 77],[KL 78],[MS 78],[Ell 79], [Ell 80]). The most important consistency requirement on a file system is that the integrity of the data structure is guaranteed. This is in contrast to a transaction processing system, where the most important integrity requirement concerns data values, and the integrity of the data structure is taken for granted.

The implementation of a file system however, must also satisfy some consistency requirements involving data values. An implementation that allows a FIND(z) to be unsuccessful while z is present in the file would be unusable. The difficulties in formulating the requirements involving data values come from time overlaps between INSERT, DELETE, FIND on the same key-value. Below we propose a formal correctness requirement on file systems. Its applicability is based only on the following assumption:

> At any instant (visualize all clocks stopped and the file system frozen)
> it is unambiguously possible to answer the question whether a given key z
> is or is not in the file system.

The file system is seen as a state machine whose behaviour is a sequence of instantaneous transitions caused by atomic actions; and for each state of the file system, we define exactly which key values are considered to be in the system. Whereas operations on the file system extend over an interval of time, the state of presence or absence of a record changes instantaneously.

Since we assume the existence of a mechanism that will issue consistent (with respect to the contents of the records) file operation requests, no synchronization on the record level is provided. That is, we consider any arbitrary sequence of operations on the records to be correct. This allows full (logical) concurrency and no constraints are placed on the relative speeds of the different processes. From the many possible concurrency protocols, we start by separating the requirements into two distinct categories. One deals with the requirements on operations on records, namely the within-bucket procedures, the other with operations that affect the locating of a bucket. Both requirement categories serve to maintain the integrity of a data structure under concurrency, the first deals with the static structure of a bucket, the other with the dynamic structure supported by a 2-level file system.

4.1 Requirements inside buckets

We consider any implementation of the within-bucket operations to be correct if the 5 requirements listed below are satisfied. We denote the time intervals of existence and non-existence of a record z by $\exists z$ and $\nexists z$ respectively.

The transitions between intervals of existence and non-existence are atomic. $\overline{f(z)}$, $\overline{i(z)}$, $\overline{d(z)}$ are abbreviations for the time intervals required by the three within-bucket operations. $\subset$ indicates inclusion.

We now can state our 5 requirements:

B1 If $\overline{f(z)} \subset \exists z$ then outcome of f(z) must be "found";

If $\overline{f(z)} \subset \nexists z$ then outcome of f(z) must be "fail".

B2 No $\overline{d(z)} \subset \exists z$.

B3 No $\overline{i(z)} \subset \nexists z$.

B4 Each transition from $\nexists z$ to $\exists z$ is overlapped by at least one $\overline{i(z)}$.

B5 Each transition from $\exists$ z to $\nexists$ z is overlapped by at least one $\overline{d(z)}$.

Requirement B1 states that if the interval of a FIND(z) is not completely contained in an $\exists$ z- or $\nexists$ z-interval (i.e. overlaps with $\overline{i(z)}$ or $\overline{d(z)}$), then outcome is unpredictable. Requirements B2 and B3 guarantee a correct result of an i(z) or d(z) operation, in other words, an intended update is always carried out. Requirements B4 and B5 state that, if several i(z) or d(z) are in the system, at least one of each will cause a change of interval.

4.2 Requirements on path to bucket

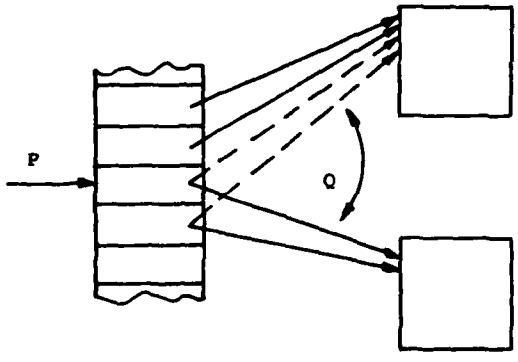
What we require is that, once a process P is in the file system, no other process Q prevents P from obtaining a relevant bucket B eventually and also, process P does not prevent any other process Q from getting eventually into any other bucket B' (B and B' not necessarily different buckets).

Looking at Fig. 4.1 we notice that getting to the desired bucket can depend on the number of paths to a bucket being constant during the locate bucket operation. Letting Δp denote a change in the number of paths to a given bucket, we can list the requirements on the path to a bucket as follows:

P1 No Δp during locate bucket interval.

P2 Each Δp can only be caused by either SPLIT, MERGE, EXTEND or REDUCE.

These two requirements guarantee that the induced operations of one process do not prevent any other process from obtaining a relevant address eventually when locating a bucket.



If Q changes these paths (due to split or merge) a concurrent process P could end up in the wrong bucket.

Fig. 4.1 No. of paths to a bucket = const. is a necessary requirement to get into the relevant bucket in all possible cases

5. A concurrent access protocol for 2-level files

The design of file systems that store data in fixed-size buckets poses two independent sub-problems:

1) Designing the structure superimposed on the collection of buckets, and the necessary operations to manipulate this structure.

2) Designing the structure of data within a bucket, and the necessary operations.

A 2-level file structure is a solution to the first sub-problem. We show that a simple protocol allows reasonably efficient concurrent access and update of the between-bucket structure. Access and update within a bucket is an independent concurrency problem that is not specific to extendible hashing. We present a simple solution that satisfies the requirements of section 4. For reasons mentioned in section 4, the notion of time for the file system might be different from that of the transaction manager. We therefore chose to implement concurrency control using locks rather than time stamps.

5.1 Locate-bucket and induced operations

One important aspect of file structures which organize the embedding space rather than the file content is the fact that we do not depend on the values of the data present in the file in order to move around the structure. This is what makes for example grid files easier to use under concurrency than tree structures. In 2-level files, a process passes by an information carrier only once while in the system, thereby it is not susceptible to changes going on "behind"•. This in turn means that we need not hold any resources (pages) while requesting others and thus no deadlock can occur. For the system to be free of deadlock, it is especially important that we do not hold directory pages while waiting for the bucket to be brought in from secondary store. Since these transfers are slow, the integrity of the data structure is in question as operations on the directory are involved. Let us look at these operations in turn.

The need for locks

The locate bucket operation consists of calculating the index of the directory entry and retrieving the bucket address from the directory. This operation makes heavy use of the directory. Therefore, we want to be certain that locate bucket operations do not collide with SPLIT, MERGE, EXTEND and REDUCE since the latter four modify the directory. In the rest of this section we will isolate a control scheme only and not concern ourselves with the implementation of the individual procedures.

The problem of extending the directory can be minimized if the directory section to be added can be prepared one entry at a time, then released in an atomic action. All processes get to the correct bucket during the transition phase, because no access path to the buckets has been changed but only the number of possible paths to a given bucket has increased.

Unfortunately there is no symmetry between EXTEND and REDUCE as far as this ease of collision avoidance is concerned. A slow process locating a bucket in an extendible hash file could use an incorrect value of d when overlapping with another process halving the directory. This would violate requirement P1. So, here we need synchronization that prevents overlapping of locate bucket operations with the reduction in size of the directory.

The other two operations which, among other things, change information in the directory are SPLIT and MERGE. Here we not only change the number of paths to a bucket but also introduce new paths. Also, a given record can, at two different instances during SPLIT or MERGE, appear in different buckets. This complicates matters especially when concurrently inserting or deleting records in a bucket involved in a SPLIT or MERGE operation. Since SPLIT and MERGE have far reaching consequences, we chose to

• Data structures and algorithms for trees have been proposed that will update a tree in a single top down pass, thereby simplifying the concurrency dependent protocols [GS 78].

synchronize them with respect to locate-bucket operations. This will automatically include all external operations. SPLIT and MERGE will, on the average, be performed with long intervals in between, which means that this synchronization does not reduce concurrency significantly. To protect locate bucket operations, the lock(s) need to be placed at the directory entry(ies). The next question then is: from where in the file system need the locks to be operated on? This is largely dependent on the trade-off between time gained through increased concurrency and time lost due to complex protocols. From Fig. 5.1 we see that when induced operations are executed, two extreme cases in terms of access path exist for a given record: 1) nothing changes (record in cross-hatched area), 2) everything changes (record in dotted area).

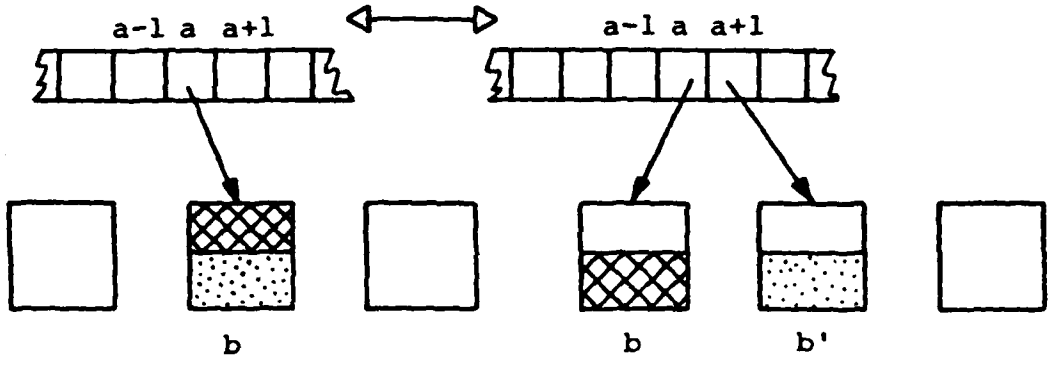


Fig. 5.1 Possible effects of induced operations

The simplest protocol will release a lock after all operations involving buckets and directory are complete. This at the risk of locking out another process unecessarily. This unnecessary lock-out can be avoided if, bucket and directory are locked separately and independently. We believe, however, that the gains made by increasing concurrency this way do not make up for the additional increase in complexity of the protocol. We then propose a simple protocol that asks only two questions before proceeding with a given access: 1) is someone else modifying the structure? or 2) how many processes have to leave the structure befor I can modify it?

Number of locks

When considering the question of granularity we can take advantage of the 2-level file structures discussed in this paper: the complexity of the directory does not change in a dynamic environment. More to the point, a user "sees" as much in a small directory as he would see in a much larger one. This permits us to investigate the effects of the locking protocol at the smallest entity: one directory entry and its associated bucket. This unit also forms the initial file and as such is one of the boundary conditions to be investigated in any case. As the file grows two choices emerge:

1) the trivial solution of controlling the entire file with one lock, or
2) introduce locks dynamically with a given granularity.

Since a change in one part of the file does not propagate to other parts of the same file (we ignore side effects inherent in a given implementation), the results valid for the smallest unit also hold when applied to a large file. The degree of concurrency can be changed with the size of the locking-granules. Fig. 5.2 depicts the idea graphically. We will not expand on questions of what degree of granularity should be chosen. Ries and Stonebraker report in [RS 77] and [RS 79] on work done concerning granularity in data bases.

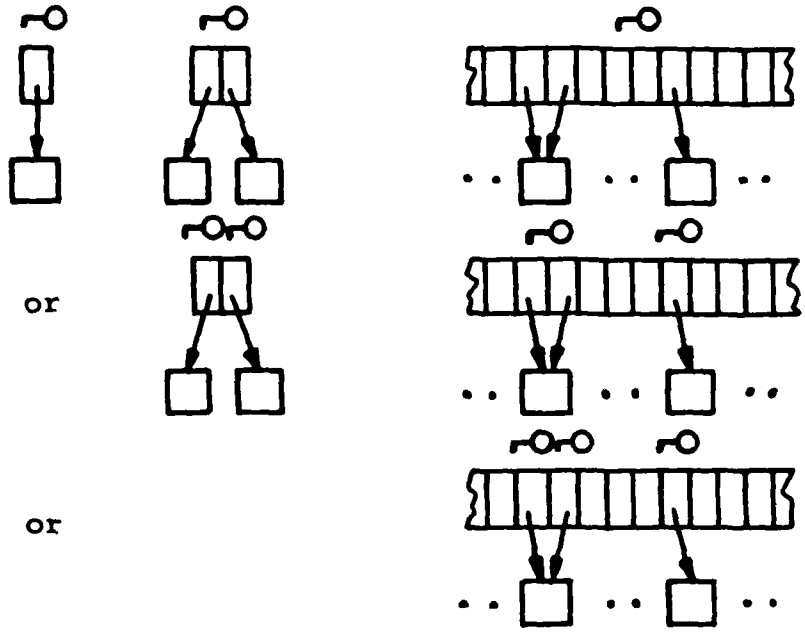


Fig. 5.2 Placing locks dynamically in a 2-level file

Characteristics of the lock

We define a global lock (global to a granule) which can be in one of two states: counting or set. Every locate bucket operation will test the lock. If the lock is in a counting state, the process will increment it and proceed. The lock is decremented once the process is finished with the bucket. If the lock is in the set state, the process will wait until the lock is released (back in counting state), then increment it and proceed. This protocol permits two modes of operation: allow a known number of processes to pass or allow no process to pass.

SPLIT and MERGE will set the lock, waiting for other operations in progress in the same bucket to complete (decrementing the lock to 0). This set mode will lock out any new locate bucket operations. Once no other operations are active, SPLIT or MERGE can proceed. If the directory is not in need of modification, the lock is released, otherwise EXTEND or REDUCE are executed and the lock released upon their completion. Since we are more interested in principles than implementation details in this paper, we will depict the lock graphically and list the external operations again with the lock mechanism built in.

The following are the lock symbols with their associated meaning.

The following compatibility table sums up the relationship between the states of the lock and attempted

actions.

state of lock	attempted action			
	increment	decrement	set	release
counting	yes	yes	yes	no
set	no	yes	no	yes

The external operations now are:

FIND(z)

1. locate_bucket(z)

2. find_within_bucket(z,B)

INSERT(z)

1. locate_bucket(z)

2. insert_within_bucket(z,B)

3. split(B)

4. double

DELETE(z)

1. locate_bucket(z)

2. delete_within_bucket(z,B)

3. merge(B)

4. halve

5.2 Within-bucket operations

Considerations of efficiency usually lead the designer to identify the logical notion of a bucket with a physical unit of storage that is transferred between central memory and secondary storage in its entirety (e.g. a disk block or a page). Since within-bucket operations require no access to secondary storage they are fast compared to the locate-bucket operations. Thus the implementation of mutually compatible within-bucket operations are uncritical. Let us consider two examples.

Single processor

An efficient implementation on a single-processor computer will queue all within-bucket operations and perform each such operation to completion before another one is started. When the desired bucket has been read into memory a within-bucket operation may well be interrupted by some I/O process, but not by any other within-bucket operations. This amounts to mutual exclusion on any one bucket and assures compatibility of the within-bucket operations in a trivial way.

Multiple processors, multiple memory banks

As an example of a more complex situation, envision a computer consisting of several processors and memory banks. Access requests by processors to memories are interleaved in an unpredictable way. A bucket may be stored in several memory banks, and several processors may be executing operations on the same bucket concurrently. For simplicity of exposition we assume that any operation we must perform on a single record (e.g. read a record, test its key and delete the entire record) is atomic. That is we have now pushed the granularity of concurrency from the bucket level down to the record level, but will not refine it further.

If records are stored in a bucket without regard to any order, then, given atomic actions, different processes working with distinct records can proceed without affecting each other. FIND and DELETE even tolerate identical records: whoever reads the record first determines the outcome (remember that another authority is concerned with the legality of sequences). With INSERT the situation is different however. Given unsynchronized processes with varying speeds, a record could be inserted twice, resulting in an illegal bucket state with respect to the duplicated key value. There exist various methods to prevent this situation from occurring. In our strive for simplicity we have chosen to restrict access to a given bucket to one INSERT only. To this end we define a lock at the bucket level. This lock is only tested and set by insert within bucket operations. We introduce a new symbol for it:


test and set


release

The following compatibility table is associated with it

state of lock	attempted action		
	find w. bkt	insert w. bkt	delete w. bkt
set	yes	no	yes
released	yes	yes	yes

A bucket is structured as an array of a fixed number of record frames. Records are stored in the bucket without regard to any order. All operations start scanning the bucket in the same direction, "from front to end", looking at one record frame at a time.

A bucket state is legal with respect to any given key value z, if and only if there is at most one entry (z). If there is, then by definition z is present; if not, it is absent. Because there is at most one entry (z), and since operations on records are atomic, the transitions between $\nexists$ z-intervals and $\exists$ z-intervals are instantaneous. The within-bucket operations can be described as follows.

find_within_bucket(z,B):

scan bucket until (z) is found:	return success;
or end of bucket:	return failure.

The assumption that looking at a record is atomic makes it trivial to verify that requirement B1) of section 4 is fulfilled. If the time interval of a find(z) operation is contained entirely within an interval of constant z, ($\exists$z or $\nexists$z), the result is correct. The verification of the other requirements is equally straight foreward.

delete_within_bucket(z,B):

scan bucket until (z) is found:	delete; update bucket occupancy; if too low send a merge signal; return "deleted";
or end of bucket:	return "not present".

Notice that only delete needs to be atomic, not including the update or signal sending.

insert_within_bucket(z,B):

scan bucket until (z) is found:	return "already present";

or end of bucket:	insert in empty slot; update bucket occupancy; return "inserted"; or request to split; return "failure";

The splitting and merging of buckets is strictly implementation dependent. We therefore will not elaborate on "request to split" and "send merge signal". Suffice it to say that the first incidence affects mostly performance with respect to time whereas the second has an influence on space utilisation.

6. Properties of the concurrent access protocol

If a bucket starts out with at most one entry (z), it can never reach a state with more than a single (z) for the following reasons. An INSERT(z,B) looks at every position in the bucket and will find (z) (if present) before end of bucket, returning "already present". If (z) is not present, the record will be inserted. Since the lock allows no more than one insert within bucket to pass, at most one record can be inserted. If a delete within bucket(z,B) is in progress at the same time then, due to the atomicity of the actions and the scanning of the entire bucket, the presence of (z) will depend on the order of arrival of the processes at the record. Find within bucket will not modify the contents of the bucket and thus will not influence any other process. Having shown that legal bucket states (at most one (z) entry) are preserved by insert within bucket, it follows directly that the requirements of section 4 are satisfied.

The protocol for external operations permits an arbitrary number of within-bucket operations to proceed concurrently inside any given bucket. It ensures that any process will either see a correct directory or no directory at all. The protocol for internal operations on the other hand protects the contents of the file.

It is certainly not difficult to see that the above simple protocol fulfills the requirements on the paths to a bucket. P2 (in this paper a requirement assumed to be met by a given implementation) guarantees us that any changes to the path can be made only after the global lock has been set, i.e. no locate bucket operations are in progress, thereby fulfilling requirement P1. In all of the above procedures we never hold resources (in particular locks) while requesting resources. Therefore deadlock cannot occur. Fig. 6.2 depicts the paths a given access to the file system can follow.

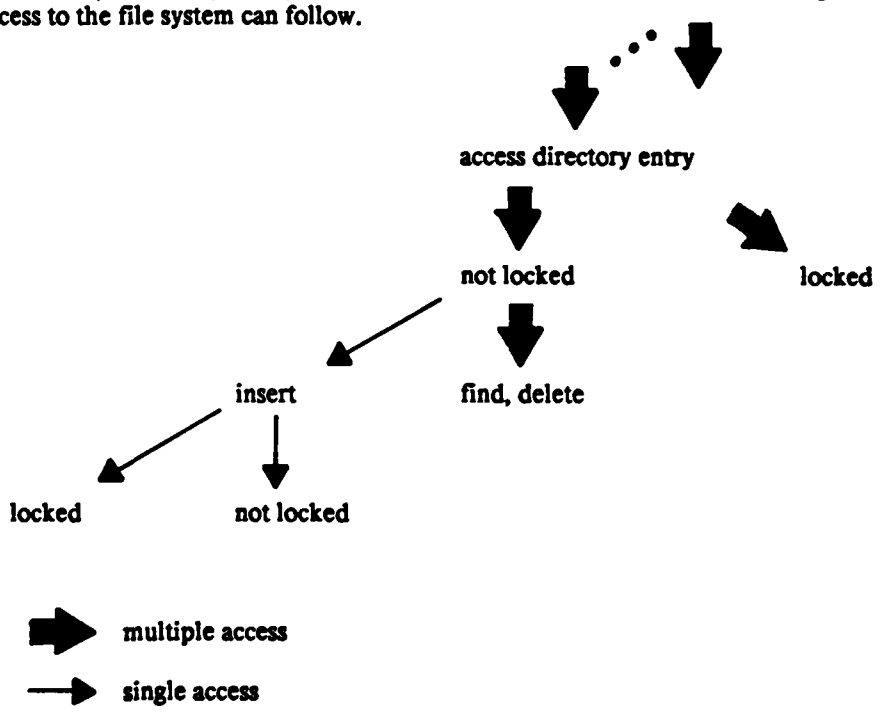


Fig. 6.2 Possible access paths to a 2-level file with the proposed protocol

Let us briefly recap some of the major point of the proposed protocol:

- Simplicity,
- No built-in, implementation-dependent constraints,
- High degree of concurrency,
- Easy control of the degree of concurrency,

- Degree of concurrency does not influence the complexity of the protocol,
- No deadlock,
- Protocol can be applied to different 2-level file structures,
- Protocol maintains 2-level structure: external & internal operations; primary & induced operations.

7. Conclusion

Studies of the effects concurrent access has on dynamic files is for all practical purposes limited to tree structures. This is not surprising, because the use of trees is widespread among file structure implementations. The protocols required to access tree structures concurrently, are complicated and costly. Any attempt to increase concurrency is overshadowed by the doubt as to whether the major increase in complexity is warranted by the additional concurrency obtained.

Correctness and consistency requirements are still largely dictated by definitions tailored to problems at the transaction level. No clear separation of the data base from the file system the DB is embedded in seems to exist. We were looking for a simple protocol allowing concurrent access in a dynamic environment and found that 2-level files provide a suitable base to implement such a system. While designing a protocol, the need for definitions of what it means for a file system to be correct under concurrency (i.e. the effects of overlapping time intervals) arose. The definitions we propose aid in the design of correct implementations of file management systems and are used to show that the simple protocol we have developed is correct. An increase in concurrency (locking granularity) can be obtained with no additional increase in complexity.

An evaluation of a given concurrency protocol can of course only be made once this protocol has been implemented and can be subjected to rigorous tests. A project currently under way at the Institut fur Informatik of ETH Zurich has as its aim the investigation of concurrent access to a grid file.

REFERENCES

[BS 77] Bayer, R., Schkolnik, M., Concurrency of Operations On B-Trees, Acta Informatica 9, 1-21 (1977).

[EGLT 76] Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L., The Notions of Consistency and Predicate Locks in a Database System, Comm. of the ACM 19, 11 (Nov. 1976) 624-633.

[Ell 79] Ellis, C., Concurrent Search And Insertion In AVL Trees, Proc. of 1979 International Conference on Parallel Processing, 1979.

[Ell 80] Ellis, C., Concurrent Search and Insertion In 2-3 Trees, Acta Informatica, 14, 63-86 (1980).

[FNPS 79] Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R., Extendible Hashing - A Fast Access Method for Dynamic Files, ACM Trans. Database Syst. Vol. 4, No. 3 (Sep. 1979), 315-344.

[GM 82] Gardarin, G., Melkanoff, M., Concurrency control principles in distributed and centralized data bases, report no. 113, INRIA, Centre de Rocqencourt, Jan. 1982.

[GS 78] Guibas, L.J., Sedgewick, R., A dichromatic framework for balanced trees, Proc. 19th ann. Symp. Foundations of computer science, IEEE 1978.

[HGLS 78] Holt, R.C., Graham, G.S., Lazowska, E.D., Scott, M.A., Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley, Reading, Mass., 1978.

[KL 78] Kung, H.T., Lehman, P.L., Concurrent Manipulation Of Binary Search Trees, Res. Rep., Carnegie Mellon Univ., Pittsburgh Pa. July 1978.

[Knu 73] Knuth, D.E., The Art Of Computer Programming, Vol. 3, Addison-Wesley, Reading, Mass., 1973.

[MS 78] Miller, R.E., Snyder, L., Multiple Access To B-Trees, in Proc. Conf. Inform. Sci. and Syst. March 1978.

[NHS 81] Nievergelt, J., Hinterberger, H., Sevcik, K.C., THE GRID FILE: an adaptable, symmetric multi-key file structure, Report no. 46, Institut fur Informatik, ETH Zurich, Dec. 1981.

[PS 81] Pohm, A.V., Smay, T.A., Computer Memory Systems, Computer Vol. 14, no. 10, (Oct. 1981), 93-110.

[RS 77] Ries, D.R., Stonebraker, M.R., Effects of locking granularity in a database management system, ACM Trans. Database Syst. Vol. 2, No. 3 (Sep. 1977) 233-246.

[RS 79] Ries, D.R., Stonebraker, M.R., Locking Granularity Revisited, ACM, Trans. Database Syst. Vol. 4, No. 2 (June 1979) 210-227.


DATE
ILME